# Modeling a Simple Verilog Cell Library

Christian Krieg

October 29, 2024

Welcome to this year's *Digital Integrated Circuits* course! Throughout this course, we will have to solve some assignments that will teach us fundamental concepts of integrated circuit design.

We will demonstrate digital integrated circuit design on a step-by-step simple example of a digital counter. As the name of this course suggests, our ultimate goal is to implement this counter as a digital integrated circuit. In the lab, we map this counter to a field-programmable gate array (FPGA) architecture, and run it on the board shown in Figure 1. The counter's functionality is executed on the FPGA. We will be able to control execution by pushing the buttons, and to visualize the counter value using the light-emitting diodes (LEDs). Because operating a hardware counter with buttons and LEDs may be inconvenient, we also will interface the hardware counter over a universal asynchronous receiver/transmitter (UART), so that we may be able to further process the counter value on a host computer. But this will all happen during the lab week at the end of the exercises.

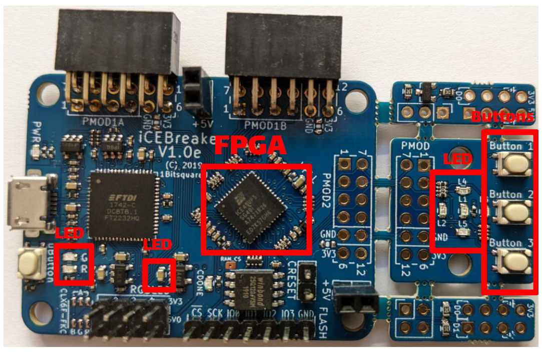

Figure 1: The *icebreaker* board, which will run our counter

Before we can bring our counter on hardware, it is important that we learn how to design digital circuits using state-of-the-art (and beyond) methods and tools, and to get a feeling for the results of our design decisions. We will draw attention on the results of the synthesis steps, which we will verify with several tools along the digital design flow.

Generally speaking, we perform the following steps:

1. Specify the counter's behavior
2. **Model (implement) the counter in a hardware description language (HDL)**
3. **Verify the counter's implementation**
4. Synthesize and optimize the counter's HDL model + verify
5. Map the counter to the target (FPGA) technology + verify
6. Generate an FPGA bitstream + verify
7. Configure the FPGA with the bitstream + verify
8. Run the counter on hardware

Our overall task is to build a synchronous counter that runs in two modes, based on a *mode* switch. The two modes are called *up* and *down*. In *up* mode, the counter increments the counter value by **5** at the rising edge of the clock signal, and in *down* mode, it decrements the counter value by **9** at the rising edge of the clock signal. The counter should implement synchronous reset behavior, such that it is set to an initial value of **-50** upon reset. The counter should never exceed a value of **235**, and should never go below a value of **-230** (the counter value sticks near the minimum/maximum value, until the counting direction changes). The counter value should never be **-11** (the counter should jump over this value by one increment/decrement).

## Wrapping up Task 3

We hope that you enjoyed to model your counter in behavioral Verilog, and that your assertions helped you to debug and verify your counter’s implementation. In case you did not come up with a working solution, we provide a working solution in Listing 1 which may be used for future tasks.

Listing 1: Sample solution for counter in Verilog (*counter.v*)

```verilog
module counter (clk, rst, mode, cnt);

   input wire clk;
   input wire rst;
   input wire mode;
   output reg signed [9:0] cnt = -10’sd50;

   reg signed [9:0] summand;

   always @* begin
      summand = 10’sd0;
      if (mode) begin
         if (cnt != -10’sd16) begin
            if (cnt <= 10’sd230) begin
               summand = 10’sd5;
            end
         end else if (cnt <= 10’sd225) begin
            summand = 10’sd10;
         end
      // mode = 0
      end else begin
         if (cnt != -10’sd2) begin
            if (cnt >= -10’sd221 ) begin
               summand = -10’sd9;
            end
         end else if (cnt >= -10’sd212) begin
            summand = -10’sd18;
         end
      end
   end

   always @(posedge clk) begin
      if (rst)
         cnt <= -10’sd50;
      else begin
         cnt <= cnt + summand;
      end
   end

   ‘ifdef FORMAL
      reg init = 1;
      always @(posedge clk) begin
         if (init) assume(rst);
         else assume(!rst);
         init <= 0;
      end

      always @(posedge clk) begin

         if (rst) begin
            assert (cnt == -10’sd50);
         end

         if (!rst) begin

            //
            // Check if counter value is never lower than MIN, larger than MAX, or
            // equal to INV
            //
            assert (cnt <= 10’sd235);
            assert (cnt >= -10’sd230);
            assert (cnt != -10’sd11);

            //
            // Check if the counter value is correctly incremented and decremented
            //

            // Counting up
            if ($past(mode)) begin
               if (!$past(rst)) begin
                  if ($past(cnt) == -10’sd16) // INV - INC, counter jumped over INV
                     assert ((cnt - $past(cnt)) == 10’sd10);
                  else if ($past(cnt) <= 10’sd235 && $past(cnt) > 10’sd230) // cnt was near MAX
                     assert ((cnt - $past(cnt)) == 0);
                  else
                     assert ((cnt - $past(cnt)) == 10’sd5);
               end
            end

            // Counting down
            if (!$past(mode)) begin
```

```
                if (!$past(rst)) begin
                   if ($past(cnt) == -10'sd2) // INV + DEC, counter jumped over INV
                      assert (($past(cnt) - cnt) == 10'sd18);
                   else if ($past(cnt) >= -10'sd230 && $past(cnt) < -10'sd221) // cnt was near MIN
                      assert (($past(cnt) - cnt) == 0);
                   else
                      assert (($past(cnt) - cnt) == 10'sd9);
                end
            end

        end
    end
   `endif
endmodule
```

As we can see, we use one addition instead of multiple adders and/or subtractors. Also, we do not host multipliers in our design. Multipliers are expensive in terms of area, and the data range is different which results in a different bit width for the output signals.

Note that we put all combinational logic into an `always @*`-block, and use *blocking* assignments (`... = ...`). Likewise, we put all sequential logic into an `always @(posedge clk)`-block, and use *non-blocking* assignments (`... <= ...`). Instead of evaluating the counter value and setting its new value in one sequential block (which may create multiple adders/subtractors), we define an intermediate signal `summand` to hold the difference to the counter's next value, which is calculated according to the counter's current value and the status of `mode`. The value of `summand` is added to the counter value at each positive edge of the `clk` signal. Note that the counter and the summand signal is `signed`, and that we explicitly declare the constants' representations to be 10-bit signed decimals. On top, we carry the counter's formal specification along with its implementation, which simplifies the verification process (less files needed).

**The take-away message here is that by choosing a suitable implementation style, we potentially save significant amounts of resources (adders and subtractors). In addition, the design is potentially clearer, and easier to maintain. Coding style matters!**

## Low-level modeling

For now, the (arithmetic) addition operation ("+") in our counter is modeled by generic *$add* cells. We provide the inputs and an obtain their sum at the outputs. We do not specify yet in detail how the sum is calculated. It is now time to define an underlying Boolean implementation of the adder in our design.

In this task, we learn to implement a simple $N$-bit carry-lookahead adder (CLA). Make yourself familiar with the concept of CLA in order to understand what it is used for. We will model our adder implementation in Verilog, and we will use technology mapping (*techmap*), a very practicable feature of *Yosys*, to substitute the generic *$add* cell by its low-level implementation.

In Verilog, functional blocks are modeled in *modules*, which may be parameterized with the construct of *parameters* (like *generics* in VHDL). A module's interface is defined using *input* and *output* directives. Like the VHDL *downto* directive, in Verilog ranges of bits are declared using brackets, with the upper bound left of a colon, and the lower-bound right of the colon (e.g., *[12:0] means from bit 12 down to bit 0*). Like VHDL, Verilog supports *generate* statements in order to instantiate multiple hardware blocks based on parameters and conditions. There are *blocking* statements (*assign*) generally used for combinational circuits, and *non-blocking* statements (<=), generally used for modeling sequential circuits. The equivalent of *signals* in VHDL are *wires* in Verilog.

In Listing 2, we can see a module definition of an *$add* cell type. The module is parameterized with the width of input and output signals, and the signedness of input signals. The parameters get default values, which are replaced by actual values when the modules are instantiated in an actual design. The module definitions also define input signals ($A$, $B$) and output ($Y$) signals. Note that while the widths of inputs $A$ and $B$ may be defined as independent values, we only consider `A_WIDTH = B_WIDTH`. Also note that the adder's output signal is one bit larger than its input signals. This is because the output carry bit is assigned to the output signal's most-significant bit.

Listing 2: Skeleton Verilog module to implement *$add* cells (*map.v*)

```
//-----------------------------------------------------------------------
//
// Maps $add cell to a carry-lookahead adder (CLA) implementation
//
module \$add (A, B, Y);

     parameter A_WIDTH = 32;
     parameter B_WIDTH = 32;
     parameter Y_WIDTH = 32;
     parameter A_SIGNED = 0;
     parameter B_SIGNED = 0;

```

```
    input [A_WIDTH-1:0] A;
    input [B_WIDTH-1:0] B;
    output [Y_WIDTH:0] Y;

    // Implement CLA functionality here

endmodule
//
//----------------------------------------------------------------------------
```

## Technology mapping

Yosys provides a very practicable feature that allows us to replace cells by their low-level implementation. It is called *technology mapping*, and implemented in the *techmap* command. *techmap* accepts a Verilog file as input which includes the mappings between cells and their implementation. When we provide *map.v* to the *techmap* command, *techmap* will look for *$add* cells, and will replace them with their low-level implementation as specified by you. The command to invoke *techmap* ist given in Listing 3.

Listing 3: *Yosys* command to replace adders by their low-level implementation

```
techmap -map map.v
```

We synthesize our counter as we did in Tasks 3, then we perform technology mapping as shown in Listing 3, and see what happens to our counter using the *show* and *stat* commands. We may write our counter to a Verilog file using the *write_verilog* command (as shown in Listing 4). We verify the functional correctness of our counter (given in Verilog) with *SymbiYosys* using our formal testbench. Listing 5 shows the *SymbiYosys* configuration file.

Listing 4: *Yosys* command to write our counter to a Verilog file

```
write_verilog counter-cla.v
```

Listing 5: *SymbiYosys* configuration to verify our counter's Verilog model (counter.sby)

```
[options]
mode prove
expect pass

[engines]
aiger suprove

[script]
read -formal tb_counter.sv counter-cla.v
verific -import tb_counter
prep -top tb_counter

[files]
counter-cla.v
tb_counter.sv
```

## Your task

Your task is to extend the skeleton module given in Listing 2 with CLA functionality for any adder bit width. Use Verilog *generate* statements. Think about how you should initialize the carry input for the adder's first bits. Use *wires* if you need to define intermediate signals. If you are unsure about the boundaries of the *generate* variable, take into consideration the output signal. If you want to access one particular bit of a signal, enclose it with brackets. For instance, if you would like to access the second bit of signal *A*, you can access it with *A[1]*.

### Submission details

Attach your extended *map.v*, as well as your generated *counter-cla.v* to your submission e-mail.